# Rpthread

Jackson Bainbridge(jdb343) and Michael Rucando (mwr72)

1. API implementation

rpthread_create: We implemented this function by first checking if the scheduler context was created or not. If no scheduler context existed, we made it and we also made a context for the current process calling rpthread_create. After this initial setup, we create the context for the thread the user wants to make, and add it to two lists. One general list for all threads and another that functions as our run queue. We then start a timer and return to the callee function.

rpthread_yield: We implemented this function by stopping the timer, setting the current thread to ready, and swapping back to the scheduler.

rpthread_exit: We implemented this function by stopping the timer, setting the running thread to finished, adjusting the return pointer, freeing the thread's stack, and swapping back to the scheduler.

rpthread_join: We implemented this function by first finding the thread were waiting for in the list of threads, and then check its status. If it is still running, we yield. When it is finished, we set the value_ptr to the return pointer of the finished thread and return.

rpthread_mutex_init: We implemented this function by setting the locked field of mutex to 0.

rpthread_mutex_lock: We implemented this function by yielding until the lock is 0. We then set the lock field of mutex to 1 and return.

rpthread_mutex_unlock: We implemented this function by setting the locked field of mutex to 0.

rpthread_mutex_destroy: Our mutex implementation did not require allocating space, therefore this function is not used.

schedule: We implemented this function by first either calling sched_rr or sched_mlfq based on what the makefile specified. We then start a timer, make the running thread the head of the runqueue, and set the context to the head of the queue.

sched_rr: We implemented this function by first changing the head of the runqueue to the next entry, setting the next entry of the old head to NULL, then adding the thread back to the queue if needed.

sched_mlfq: We implemented this function the same as sched_rr, except instead using addToMlfq.

timerStart: We implemented this function by creating itimervals for the setitimer function based on the timeslice specified by the makefile.

timeStop: We implemented this function by creating itimervals with all values set to 0.

sigHandler: We implemented this function by lowering the thread's priority in mlfq, setting the thread to ready, and swapping back to the scheduler.

addThread: We implemented this function to add a thread to a rr runqueue. It does this by setting the thread to the rqhead if the runqueue does not exist, or adding the thread to the end of the runqueue,

addArray: We implemented this function to add a thread to the global list of created threads. This works in the same way as addThread.

addToMlfq: We implemented this function to add a thread to the mlfq runqueue. It does this by either setting the thread to the head of the runqueue, or adding the thread at the end of its priority ie: walking back through the list until it's priority is greater than the next entry's.

printQueue & arrayPrint: These were simply testing methods

1. Benchmark Results

Parallel_cal

With RR

- With default threads time is 2360 microseconds, sum is correct.
- With 50 threads time is 2362 microseconds, sum is correct.

With MLFQ

- With default threads time is 2358 microseconds, sum is correct.
- With 50 threads time is 2386 microseconds, sum is correct.

Vector_multiply

With RR

- With default threads time is 12 microseconds, result is correct.
- With 50 threads time is 35 microseconds, result is correct.

With MLFQ

- With default threads time is 12 microseconds, result is correct.
- With 50 threads time is 38 microseconds, result is correct.

External_cal

With RR

- With 6 threads time is 5949 microseconds, result is correct.
- With 50 threads time is 5945 microseconds, result is correct.

With MLFQ

- With 6 threads time is 5981 microseconds, result is correct.
- With 50 threads time is 5966 microseconds, result is correct.

2. Benchmark Result Analysis

The following information is a comparison of the times from our rpthread library and the pthread library

Parallel_cal

With RR

- With default threads: pthread = 619 microseconds, rpthread = 2360 microseconds.
- With 50 threads: pthread = 363 microseconds, rpthread = 2362 microseconds.

With MLFQ

- With default threads: pthread = 627 microseconds, rpthread = 2358 microseconds.
- With 50 threads: pthread = 384 microseconds, rpthread = 2386 microseconds.

Vector_multiply

With RR

- With default threads: pthread = 166 microseconds, rpthread = 12 microseconds.
- With 50 threads: pthread = 296 microseconds, rpthread = 35 microseconds.

With MLFQ

- With default threads: pthread = 113 microseconds, rpthread = 12 microseconds.
- With 50 threads: pthread = 312 microseconds, rpthread = 38 microseconds.

External_cal

With RR

- With default threads: pthread = 3791 microseconds, rpthread = 5949 microseconds.
- With 50 threads: pthread = 2355 microseconds, rpthread = 5945 microseconds.

With MLFQ

- With default threads: pthread = 3857 microseconds, rpthread = 5981 microseconds.
- With 50 threads: pthread = 2401 microseconds, rpthread = 5966 microseconds.